Tehnici de Optimizare
Laborator 1
– Introducere –

## 1 Scipy.optimize

Pachetul SciPy *optimize* (`https://docs.scipy.org/doc/scipy/tutorial/optimize.html`) conţine algoritmi pentru rezolvarea problemelor de optimizare neliniară (neconvexe). Pachetul unifică implementări ale solver-elor pentru probleme neliniare, programare liniară, probleme de regresie (Cele-Mai-Mici-Pătrate) neliniare şi constrânse, calcul de rădăcini şi aproximări de curbe etc.

Funcţia principală de minimizare a unei funcţii scalare de una sau mai multe variabile este *minimize*.

```
scipy.optimize.minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None,
bounds=None, constraints=(), tol=None, callback=None, options=None)
```

Principalii parametri :

- **fun**: Funţia obiectiv de minimizat.

  ```
  fun(x, *args) -> float
  ```

  unde $x$ este un array $1-D$ cu forma $(n,)$ şi *args* este o colecţie de parametri fixaţi (constante).
- **x0**: Punct iniţial. Tablou de elemente reale de dimensiune $(n,)$, unde $n$ este numărul de variabile.
- **args**: Argumente transmise funcţiei obiectiv si derivatelor ei.
- **method**: Algoritmul de optimizare. Exemple: 'Nelder-Mead', 'Powell', 'CG' etc.
- **jac**: Funcţie de definire a Jacobianului
- **hess**: Funcţie de definire a matricei Hessiene

**Exemplu.** Fie funcţia pătratică:

$$f(x) = 2x_1^2 + 2x_2^2 + 3x_1x_2.$$

Soluţia problemei $\min_x f(x)$ este $x^* = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Redăm codul de rezolvare folosind `minimize`:

```
x0 = (5,5)
A = np.ones((2,2)) + np.eye(2)
fun = lambda x : x.T@A@x + x[0]*x[1]
res = minimize(fun,x0,method='Nelder-Mead')
print(res.x)
```

Soluţia returnată de algoritmul simplex Nelder-Mead este `res.x` $= \begin{bmatrix} -1.56554047e-05 \\ 2.32754122e-05 \end{bmatrix}$.

**Exemplu.** Fie problema de optimizare:

$$\begin{aligned} \min_{x \in \mathbb{R}^2} \quad & (x_1 - 3)^2 + (x_2 - 2)^2 \\ \text{s.t.} \quad & \|x\|_1 \leq 1.5 \end{aligned}$$

```
def f(x):
      return np.sqrt((x[0] - 3)**2 + (x[1] - 2)**2)

def constraint(x):
      return np.atleast_1d(1.5 - np.sum(np.abs(x)))

optimize.minimize(f, np.array([0, 0]), method="SLSQP",
                  constraints={"fun": constraint, "type": "ineq"})
```

```python
import cvxpy as cp

# Create two scalar optimization variables.
x = cp.Variable()
y = cp.Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = cp.Minimize((x - y)**2)

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve()  # Returns the optimal value.
print("status:", prob.status)
print("optimal value", prob.value)
print("optimal var", x.value, y.value)
```

```
status: optimal
optimal value 0.999999999761
optimal var 1.00000000001 -1.19961841702e-11
```

## 2 CVXPY

CVXPY este un limbaj de modelare (open source: `https://www.cvxpy.org/`) în Python pentru rezolvarea problemelor de optimizare convexă. Considerăm problema:

$$\begin{aligned} &\min_{x,y} \ (x-y)^2 \\ &\text{s.l. } x+y=1, x-y \geq 1. \end{aligned}$$

Codul de modelare şi rezolvare a problemei este:
În urma execuţiei se returnează:

## 3 Filtrarea unei imagini folosind CVXPY

Ca exerciţiu vom testa eliminarea zgomotului din componenţa unei imagini "corupte" folosind programarea convexă. Presupunem cunoaşterea pixelilor afectaţi de zgomot din imaginea coruptă. Notăm cu $C$ mulţimea poziţiilor $(i,j)$ a pixelilor neafectaţi de zgomot.

O imagine alb-negru (*grayscale*) este reprezentată numeric de o matrice $A$ a intensităţilor de gri, i.e. $[0,255]$. În imaginea "coruptă" cunoaştem un set de poziţii ale pixelilor neafectaţi de zgomot. Sarcina noastră este recompletarea imaginii prin calcularea valorilor pixelilor lipsă.

Reconstrucţia se poate realiza prin minimizarea variaţiei totale a imaginii, supusă la constrângerea de menţinere a valorilor pe pixelii cunoscuţi. Una dintre formulări modelului de optimizare este:

$$\begin{aligned} &\min_{U \in \mathbb{R}^{m\times n}} TV(U) := \sum_{i=1}^{m}\sum_{j=1}^{n} \left\| \begin{bmatrix} U_{i+1,j} - U_{i,j} \\ U_{i,j+1} - U_{i,j} \end{bmatrix} \right\|_2 \\ &\text{s.l. } \ U_{ij} := A_{ij} \quad \forall (i,j) \in C. \end{aligned} \tag{1}$$

Încărcaţi imaginea originală şi pe cea coruptă:

```
import matplotlib.pyplot as plt
import numpy as np
import cvxpy as cp

img_orig = plt.imread("imag.png")[:,:,0]
img_corr = plt.imread("imag_corrupted.png")[:,:,0]
rows, cols = img_orig.shape
```

Realizăm o hartă a pixelilor neatinşi din imaginea coruptă (`known` este 1 dacă pixelul este cunoscut, 0 dacă pixelul este corupt):

```
known = np.zeros((rows, cols))
for i in range(rows):
  for j in range(cols):
     if img_corr[i, j] == img_orig[i, j]:
          known[i, j] = 1
```

Afişam cele două imagini:

```
fig, ax = plt.subplots(1, 3, figsize=(10, 5))
ax[0].imshow(img_orig, cmap='gray')
ax[0].set_title("Original Image")
ax[0].axis('off')
ax[1].imshow(img_corr, cmap='gray');
ax[1].set_title("Corrupted Image")
ax[1].axis('off');
```

Pentru rezolvarea problemei scriem minimizarea operatorului de variaţie totală în CVX:

```
U = cp.Variable(shape=(rows, cols))
obj = cp.Minimize(cp.tv(U))
constraints = [cp.multiply(known, U) == cp.multiply(known, img_corr)]
prob = cp.Problem(obj, constraints)
```

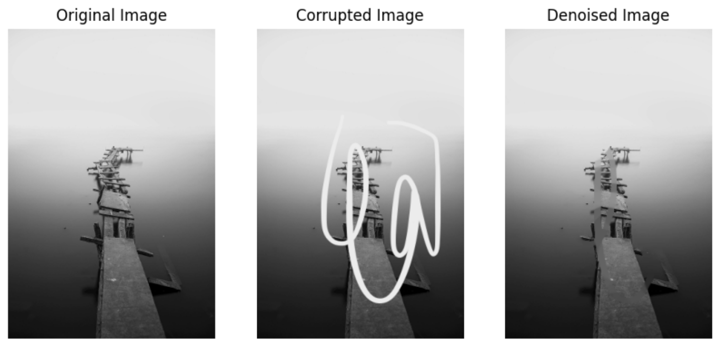
Variabila $U$ reprezintă variabila de căutare a problemei. Funcţia *cp.Minimize* defineşte obiectivul minimizării (funcţia *cp.tv(U)* implementează funcţia obiectiv din (1)). Constrângerile asigură păstrarea pixelilor (de pe poziţiile nealterate) din variabila $U$ la aceeaşi valoarea cu pixelii nealteraţi din imaginea coruptă.

```
prob.solve(verbose=True, solver=cp.SCS)
print("optimal objective value: {}".format(obj.value))
```

Funcţia *prob.solve* apelează solver-ul SCS pentru calculul soluţiei. Afişarea finală a imaginii filtrate:

```
img_denoised = U.value

ax[2].imshow(img_denoised, cmap='gray');
ax[2].set_title("Denoised Image")
ax[2].axis('off');
plt.show()
```




## 4 Probleme propuse

**P1**. Alegeţi o imagine oarecare alb-negru. Adăugaţi ”zgomot” folosind orice program de prelucrare imagini (e.g. Paint din Windows). Echivalentul numeric al imaginii alterate îl vom nota cu $Y \in \mathbb{R}^{m\times n}$

a) Urmăriţi paşii de mai sus pentru a elimina zgomotul şi a evalua rezultatul.

b) Consideraţi următoarea variantă a problemei:

$$\min_U \frac{1}{2}\|U - Y\|_F^2 + \rho G(U)$$

unde $G(U) := \sum_{i=1}^{m}\sum_{j=1}^{n} \left\| \begin{bmatrix} U_{i+1,j} - U_{i,j} \\ U_{i,j+1} - U_{i,j} \end{bmatrix} \right\|_2^2$. Echivalaţi problema cu sistemul rezultat din condiţiile de optimalitate de ordin I. Aflaţi soluţia $U^*$ pe baza rezolvării acestui sistem cu CVXPY.

**P2**. Fie problema de optimizare neconstrânsă (Rosenbrock):

$$\min_{x\in\mathbb{R}^2} \ 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

a) Aplicaţi functia *minimize* pentru a rezolva problema. Variaţi punctul iniţial şi algoritmul de rezolvare.

b) Creaţi o figură care indică graficul funcţiei obiectiv şi punctul de optim găsit

*Indicatii:* Punctul de minim este $(1, 1)$.

**P3**. Cantităţile $a_1, a_2, \cdots, a_m$ ale unui anume produs vor fi transportate din fiecare $m$ locaţii de plecare către $n$ destinaţii şi primite în fiecare dintre aceste destinaţii în cantităţile $b_1, b_2, \cdots, b_n$, respectiv. Livrarea de la locaţia-origine $i$ la locaţia-destinaţie $j$ are un cost $c_{ij}$. Se urmăreşte determinarea distribuţiei optime a stocurilor prin calcularea fiecărei cantităţi $x_{ij}$ de livrat între fiecare dintre perechile de locaţii origine-destinaţie $i = 1, 2, \cdots, m; j = 1, 2, \cdots, n$. Distribuţia optimă va respecta constrângerile de stocare şi va minimiza costul de transport.

a) Modelaţi problema printr-o problemă de programare liniară.

b) Generaţi date aleatoare pentru această problemă. Rezolvaţi modelul folosind funcţia *linprog*(Scipy.optimize) şi, separat, folosind pachetul CVXPY.

**P4**. Generaţi date aleatoare $(Q, A, c, b)$ pentru problema:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \ & \frac{1}{2}x^T Q^T Q x + c^T x \\ \text{s.t. } & Ax = b, x \geq 0. \end{aligned}$$

Rezolvaţi modelul folosind Scipy.optimize şi, separat, folosind pachetul CVXPY.

# 5 Appendix

Problemele de programare neliniară se descriu prin determinarea minimului (sau maximului) funcţiei obiectiv (criteriu), sub constrângerile variabilelor de decizie. Considerând funcţia obiectiv $f : \mathbb{R}^n \to \mathbb{R}$ definim problema:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \ & f(x) \qquad\qquad (NLP) \\ \text{s.t. } & g(x) \leq 0, \ \ h(x) = 0, \\ & Ax = b, \ \ Cx \leq d. \end{aligned}$$

Mulţimea fezabilă în care se face căutarea punctului de optim $x^*$ este definită de:

$$Q = \{x \in \mathbb{R}^n : \ g(x) \leq 0, \ \ h(x) = 0, Ax = b, \ \ Cx \leq d\}.$$

Problema (NLP) este convexă dacă funcţia obiectiv $f$ şi mulţimea fezabilă $Q$ sunt convexe.

**Programare Liniară**. Problemele de Programare Liniara (LP) sunt convexe şi sunt definite ”standard”:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \ & c^T x \qquad\qquad (LP) \\ \text{s.t. } & Ax = b, \\ & x \geq 0 \end{aligned}$$

unde observăm o funcţie obiectiv liniară şi constrângeri liniare de egalitate şi inegalitate, i.e. $c \in \mathbb{R}^n, A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$.

**Programare Pătratică**. Problemele de Programare Pătratică (QP) sunt problemele cu cost pătratic şi constrângeri liniare de forma:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \ & \frac{1}{2}x^T H x + q^T x \qquad\qquad (QP) \\ \text{s.t. } & Ax = b, \\ & Cx \leq d \end{aligned}$$

unde $A \in \mathbb{R}^{m\times n}, C \in \mathbb{R}^{p\times n}$. Dacă $H \succeq 0$, atunci problema (QP) este convexă.